Scalable Delivery of Stream Query Result

Yongluan Zhou
University of Southern Denmark
zhou@imada.sdu.dk

Ali Salehi   Karl Aberer
EPFL, Switzerland
{ali.salehi, karl.aberer}@epfl.ch

ABSTRACT

Continuous queries over data streams typically produce large volumes of result streams. To scale up the system, one should carefully study the problem of delivering the result streams to the end users, which, unfortunately, is often overlooked in existing systems. In this paper, we leverage Distributed Publish/Subscribe System (DPSS), a scalable data dissemination infrastructure, for efficient stream query result delivery. To take advantage of DPSS's multicast-like data dissemination architecture, one has to exploit the common contents among different result streams and maximize the sharing of their delivery. Hence, we propose to merge the user queries into a few representative queries whose results subsume those of the original ones, and disseminate the result streams of these representative queries through the DPSS. To realize this approach, we study the stream query containment theories and propose efficient query grouping and merging algorithms. The proposed approach is non-intrusive and hence can be easily implemented as a middleware to be incorporated into existing stream processing systems. A prototype is developed on top of an open-source stream processing system and results of an extensive performance study on real datasets verify the efficacy of the proposed techniques.

1. INTRODUCTION

The results of stream queries are typically in the form of continuous streams, whose delivery from the processing server to the end users is bandwidth consuming and hence should be carefully handled. Unfortunately, this problem is often overlooked in existing systems, which often assume the result streams are directly sent to the users from the server. Such an architecture cannot scale to a large population of users. Scaling up stream query result delivery is an important stepping stone towards massive stream query processing.

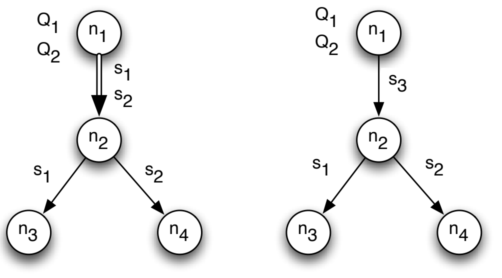
1.1 Motivating Scenario

The work in this paper is initiated to resolve a performance issue that we faced for deploying a stream processing system shared by environmental scientists from multiple institutions in the Swiss-Experiment project (`http://www.swiss-experiment.ch`). In the project, environmental scientists are deploying a number of sensor stations to study the environmental changes and to provide alerts if needed (e.g., avalanche alert, etc). An on-going effort of the project is to share the sensor data to other scientists and to the public over the world via Microsoft's SensorMap [25]. This potentially requires the processing of a huge number of real-time stream queries. Delivering their results to the massive end users is one of the challenging problems of this platform.

To solve this issue, we propose to leverage the existing scalable data dissemination infrastructure, namely distributed pub/sub system (DPSS) [8], for query result delivery. A DPSS is typically backed by a number of brokers. Users express their data interest as user subscriptions which are propagated to the brokers. The data sources need not keep track of all the end users, and instead they only push the messages to their neighboring brokers, which cooperate with other brokers to disseminate the messages to the end users. Messages are routed within the network based on their content instead of explicitly specified destinations. With such a loosely coupled architecture, DPSS is shown to be scalable to a large number of users.

One can adapt a DPSS to disseminate the query result streams as follows. In a stream processing system, one query result stream is generated for each query. Hence, a unique identifier can be assigned to each query result stream. Then a user's subscription (i.e. the user's data interest) can be composed by specifying this unique identifier to retrieve the query result stream. However, such a straight-froward approach is inefficient and involves large communication overhead. This is because the result streams could have overlap contents. Disseminating these streams individually incurs many duplicate data transfers.

To illustrate the problem, Table 1 lists a few queries specified using CQL [31]. These queries are extracted and simplified from the typical snow drift monitoring tasks of the scientists. Consider the join queries, Q_1 and Q_2, presented in Table 1. We can see the overlaps in the result streams generated for Q_1 and Q_2. Consider an overlay network structure depicted in Figure 1(a). Suppose nodes n_3 and n_4 post two



(a) Non-Share (b) Share

Figure 1: Result stream delivery

Table 1: Example Queries

Q_1:	SELECT	S2.*
	FROM	Station1 [Range 30 Minutes] S1, Station2 [Now] S2
	WHERE	S1.snowHeight > S2.snowHeight
Q_2:	SELECT	S1.snowHeight, S1.timetamp, S2.snowHeight, S2.timestamp
	FROM	Station1 [Range 1 Hour] S1, Station2 [Now] S2
	WHERE	S1.snowHeight > S2.snowHeight
Q_3:	SELECT	S2.*, S1.snowHeight, S1.timestamp
	FROM	Station1 [Range 1 Hour] S1, Station2 [Now] S2
	WHERE	S1.snowHeight > S2.snowHeight

queries Q_1 and Q_2 respectively and node n_1 is responsible for processing them. Using traditional techniques, their result streams, s_1 and s_2, are transmitted separately as shown in Figure 1(a). Hence the overlapping contents of s_1 and s_2 are transmitted twice over the link between n_1 and n_2 (n_2 is involved here because it is the neighboring broker of n_1 in the DPSS).

Note that existing multi-query optimization techniques, such as [24], cannot solve this problem. For instance, one shared join operator can be created for the above two queries. However this join operator still generates two separate result streams for the aforementioned queries respectively.

To resolve this issue, we have to send one result stream s_3 to n_2, which is the superset of both s_1 and s_2, and "split" s_3 into two separate streams s_1 and s_2 at node n_2. This approach is illustrated in Figure 1(b). One can implement this approach by re-engineering a "specialized" stream processing engine to generate one result stream for multiple queries. However, such an intrusive approach is undesirable as it requires complex "low-level" software development and tightly coupled interactions between the processing engine and the overlay network.

This paper proposes a query reformulation approach, which is relatively simple and easy to be implemented as a middleware between an existing stream processing engine and a DPSS. In our approach, for a group of queries that have overlapping results, the system composes a new query Q, called representative query, that contains all the queries in its group, i.e. the result of Q is a superset of the result of its group members. For example, instead of submitting Q_1 and Q_2 individually, we create a new query Q_3 listed in Table 1, which contains Q_1 and Q_2, and we submit Q_3 to the processing engine at n_1. The result stream s_3 will be "split" at n_2 by using the filtering mechanism within the DPSS.

1.2 Contributions

In summary, we have made the following contributions:

• We study the problem of stream query containment with a focus on window predicates, which do not exist in traditional SQL queries. The containment theorems are not limited to this work and could benefit future studies on stream query processing, such as multi-query optimization.

• Based on the containment theorems, we propose the query merging algorithm, which is meant to be simple in order to be executed efficiently at run time.

• We consider the situation that queries are inserted and terminated frequently and propose an efficient query grouping optimization and re-optimization mechanism. Queries are organized into a multi-tree data structure based on their containment relationship. This enables the adaptation algorithm to efficiently determine whether it is necessary to re-optimize the current grouping.

• A prototype system is implemented on top of an open source stream processing system. Extensive experiments running on real datasets show that our approach is both efficient and effective.

1.3 Layout

The rest of this paper is organized as follows. Related work is first reviewed in Section 2. Then Section 3 presents the assumptions and the system model for this paper. Section 4 addresses the problem of how to generate the representative queries and the user subscriptions. Query grouping and its maintenance issues are addressed in Section 5. Section 6 provides an extensive performance evaluation study to verify our approach. Finally, Section 7 concludes the paper with a discussion on the future work.

2. RELATED WORK

This paper is mainly related to the research activities in two areas: data stream processing systems and distributed publish/subscribe systems.

2.1 Data Stream Processing Systems

Stream processing has attracted much attention from the database community due to its vast applicability. There exist many efforts to enhance the scalability of these systems. One direction is to exploit the sharing computation among the queries. For instance, Madden et al. [24] and Chen et al. [14] proposed to share the join and filter operations among multiple queries and Arasu et al. [4], Zhang et al. [33] and Krishnamurthy et al. [22] studied the computation sharing of sliding-window aggregates. While these methods are effective in making the computation resources scalable, they do not consider the data communication aspect. This paper complements the literatures by exploiting the sharing among queries to minimize the cost of result delivery. Our approach can co-exist with existing computation sharing algorithms within a stream system due to its non-intrusiveness.

There are also literatures about employing a number of distributed servers to share processing load. The authors of [3] studied the problem of how to place the query operators to the widely distributed servers. In [32], the operator placement problem in a locally distributed system is investigated. Our approach is also complementary to these type of efforts. It can be used to disseminate the result streams of the queries/operators allocated to a processing server to its downstream destinations in a distributed stream processing system.

We have recently proposed a query allocation scheme in a companion paper [34]. In [34], we employed a distributed publish/subscribe system to disseminate the stream data from the data sources to the processing servers and focused on optimizing the allocation of the queries to the servers in order to achieve both load balancing and communication minimization. On the other hand, this paper solves the other aspect of the problem by leveraging the pub/sub system to deliver the query results and focuses on merging queries within each individual node.

Another direction in scalable stream processing is to shed the excessive workload when the data arrived much faster than what the system can handle. Reference [30] presented an input tuple shedding strategy to maximize the query result quality. Authors in [6] proposed another tuple shedding strategy to minimize the loss of aggregate accuracy that would be incurred by the shedding. While one can adopt a similar strategy when the server runs out of bandwidth to deliver query results, it sacrifices the accuracy of the results. Our approach tries to adopt a better result delivery architecture, namely DPSS, to avoid (or minimize) the occurrence of such cases.

2.2 Distributed Pub/Sub Systems

Many literatures have been focused on the scalability of distributed pub/sub systems. For instance, efficient matching of events with subscriptions within a broker is studied in [2]. Authors in [8, 10, 9] presented the architecture design of a DPSS with a number of widely distributed brokers. Our approach leverages these existing efforts to enhance the scalability of a stream processing system. Detecting subscription containments [27, 19] and merging subscriptions [16] has also been explored in traditional DPSSs. However, the subscriptions considered are only simple selection predicates and hence cannot be applied to our problem.

There are also very recent efforts to extend pub/sub systems to support more complex subscription types, such as range-MIN (or MAX or DISTINCT) in [12], select-natural-join in [13], and XML stream filtering [17]. However, these literatures focus on specific query types; it lacks a systematic study of general SQL-like queries. Furthermore, operators such as window joins and window aggregates, which are heavily used in many stream applications, have not been discussed in previous work.

3. PRELIMINARIES

The whole system consists of a stream processing server, a DPSS infrastructure, and a number of end users. It is assumed that the end users have limited computing power and can only perform simple operations such as projection and selection. Complex operations like window joins and window aggregates can only be processed at the processing server. Furthermore, to loosen the coupling between the server and the DPSS, we assume the server has little knowledge of the internal overlay structure of the DPSS.

3.1 DPSS

A subscription in the DPSS is a triple $\langle \mathcal{S}, \mathcal{P}, \mathcal{F} \rangle$. $\mathcal{S}$ is a set of stream names, which indicates the streams that are of interest to the subscriber. Only data from these streams would match the subscription. In our system, a unique stream name is assigned to each result stream of a query running in the processing engine. Hence the users can retrieve their query results by subscribing to the corresponding result streams. $\mathcal{P}$ specifies a few selected attributes from the streams in $\mathcal{S}$ that are of interest to the subscriber. Finally, $\mathcal{F}$ is a set of filters over the streams within $\mathcal{S}$. Data from the streams that satisfy these filters will be sent to the subscriber.

In the DPSS, subscriptions are forwarded from the subscribers to the data source. On the way of the forwarding, an intermediate node aggregates all the subscriptions that are received before forwarding to its upper stream neighbor(s). Furthermore, each node will build their own routing table based on the subscriptions it has. Upon receiving a message, the routing table is used to determine which downstream neighbor(s) the message should be sent to. If the message matches any subscription forwarded from a neighbor, it will be delivered to that neighbor. It can be seen that, even if there are more than one subscribers behind that neighbor interested in the same message, it will be sent only once.

3.2 Continuous Stream Queries

An SPJ query Q is assumed to contain the following components:

(1) $Strm(Q)$: the set of streams involved by the query Q, $\{s_1, s_2, \cdots\}$, which typically appear in the FROM clause of the SQL string.

(2) $Window(Q)$: a set of window predicates $\{w_1, w_2, \cdots\}$, one for each stream in $Strm(Q)$. For brevity, this paper only discusses time-based sliding window, while other types of window can be treated similarly. The value of a time stamp is assumed to be a non-negative integer. A window predicate is defined as follows:

Definition 3.1. *A window predicate w_i takes an input stream s_i and three non-negative integer parameters:*

- $begin_i \in [0, +\infty)$*: the starting time of the query*
- $interval_i \in [0, +\infty)$*: the interval of the window*
- $slide_i \in [1, +\infty)$*: the sliding step of the window*

It defines a temporal relation $\mathcal{R}(\tau) = \{t | t \in s_i \;\; \& \;\; 0 \leq \tau - t.timestamp < interval_i\}$ at each time instance of $\tau = begin_i + n \cdot slide_i$, where n is a non-negative integer.

For example, in Figure 2, the window predicate on s_1 defines a temporal relation on each time instance $\tau \in 0, 10, 20 \cdots$. For instance, at $\tau = 20$ and $\tau = 30$, it defines two temporal relation containing tuples within the rectangle drawn in Figure 2(a) and Figure 2(b) respectively.

(3) $Pred(Q)$: the predicate specified by Q, which appear in the WHERE clause of the SQL string. $Pred(Q)$ is assumed to be in the disjunctive normal form: $\sigma_1 \vee \cdots \vee \sigma_i \vee \cdots \vee \sigma_n$, where σ_i is the conjunction of one or more atomic predicate. An atomic predicate could be in one of the forms $attr.op.value$ and $attr1.op.attr2$ and involves the attributes from one (a selection predicate) or two streams (a join predicate).

(4) $Attr(Q)$: the set of attributes selected by Q.

An aggregate query Q takes one input stream, which could be the output of a SPJ query. In summary, Q contains the following components:

(1) *Strm*(Q): the input stream of Q;

(2) $Window(Q)$: the window predicate defined on the input stream;

(3) *Groupby*(Q): the set of grouping attributes;

(4) *Agg*(Q): the set of aggregate functions;

(5) $Having(Q)$: the filters applied over the groups;

(6) *Attr*(Q): the set of selected attributes.

3.3 Approach Overview

The server partitions the queries into a number of groups such that queries inside each group have overlapping results. For each group, one representative query Q that contains all the member queries Q_i is generated.Only the representative queries are inserted into the underlying query engine and result streams of these queries are pushed into the DPSS.

To allow the users to retrieve the query result streams of the individual queries, subscriptions are also generated and sent to the users. The users register these subscriptions to the DPSS, which efficiently delivers the result streams back to the users.

In the example presented in Section 1, the following two subscriptions are sent to n_2 by n_3 and n_4 respectively:

- p_1: $\mathcal{S} = \{s_3\}, \mathcal{P} = \{S2.*\}, \mathcal{F} = \{-30(minute) \leq S1.timestamp - S2.timestamp \leq 0\}$.
- p_2: $\mathcal{S} = \{s_3\}, \mathcal{P} = \{S1.snowHeight, S1.timetamp, S2.snowHeight, S2.timestamp\}, \mathcal{F} = \{\}$

Tuples that pass p_1 are sent to n_3 and those that pass p_2 are sent to n_4.

4. QUERY MERGING

This section first studies the stream query containment problem and then presents the query merging algorithms based on query containment theorems developed. Finally it presents the algorithm to generate the subscriptions for the users to retrieve the result from the DPSS.

4.1 Stream Query Containment

Query containment and equivalence is a fundamental problem which has been extensively studied in the literature. For example, [11] and [29] studied the conjunctive select-project-join queries and union thereof; [15] and [26] discussed the aggregate queries; [21] studied queries with arithmetic comparison predicates; [7] investigated problems of recursive queries. We, however, need to extend these techniques to the continuous stream query context. On the other hand, some related literatures studied the use of views to answer user queries [18]. This direction addresses how to rewrite a query such that the given views of the underlying relations can be utilized to answer the original query. Our work is the other way round. We have to compose a "view" of the streams that can be utilized to answer multiple queries using the simple filtering mechanism in a DPSS.

First of all, we have to extend the query containment and equivalence definition of traditional queries to continuous stream queries. Traditionally, query containment and equivalence under set semantics is defined as follows.

DEFINITION 4.1. *A query Q_1 is contained by another query Q_2, denoted by $Q_1 \sqsubseteq Q_2$, if for all database instances D, $Q_1(D)$ is a subset of $Q_2(D)$, i.e. $Q_1(D) \subseteq Q_2(D)$, where $Q_i(D)$ is the result of evaluating Q_i over D. Q_1 and Q_2 are equivalent if $Q_1 \sqsubseteq Q_2$ and $Q_2 \sqsubseteq Q_1$.*

However, continuous stream queries generate result continuously and hence this traditional definition is no longer applicable. To address this problem, we extend the definition as follows. First it is assumed that there is an application discrete time domain $\mathcal{T}$ where the timestamps of the input stream data are drawn from. We denote the temporal result data set of a query Q evaluated on a stream instance S at the time instance $\tau \in \mathcal{T}$ be $Q(S, \tau)$, which is the result of evaluating Q over all the data from S with timestamps smaller or equal to τ. Furthermore, let S be the whole set of streams. We have the following definition.

DEFINITION 4.2. *A continuous stream query Q_1 is contained by another continuous stream query Q_2, denoted by $Q_1 \sqsubseteq Q_2$, if for all stream instances S, $Q_1(S, \tau) \subseteq Q_2(S, \tau)$ at any application time instance τ. Q_1 and Q_2 are equivalent if $Q_1 \sqsubseteq Q_2$ and $Q_2 \sqsubseteq Q_1$.*

Now the problem is how to determine the containment relationship between two continuous stream queries. The major difference between continuous stream queries and traditional database queries is the introduction of window semantics. Note that if all window predicates in a continuous stream query have an infinite time interval, then the two containment problems are equivalent. This paper assumes that there is an approach to determine containment relationship between two traditional database queries, and develops the theorems to deal with the window predicates.

Furthermore, it is assumed that all the window predicates in a single query have a common sliding step and a common starting time. This covers most real application scenarios and simplifies the query merging algorithms. (Note that the sliding steps and starting times of different queries could be different.)

First, we have the following lemma stating the conditions that two tuples could be joined in a window-based join operator.

LEMMA 4.1. *For a query with only a window-based join operation of two streams s_1 and s_2 with window sizes of $interval_1$ and $interval_2$ respectively and a common sliding step "slide" and a common query starting time "begin", two tuples t_1 from s_1 and t_2 from s_2 can generate a join result tuple t if and only if all the following conditions are true:*

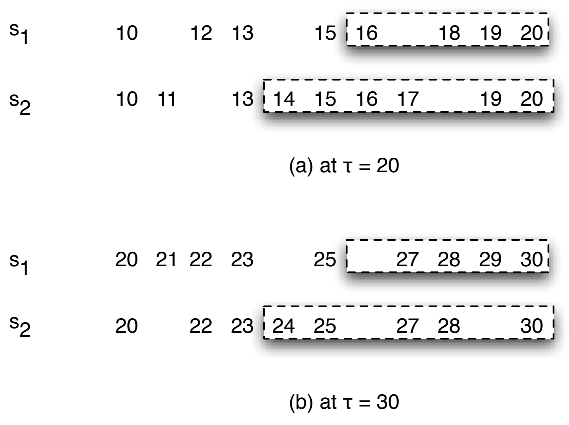
(1) they satisfy the join predicates;

(2) $-1 \cdot interval_1 \leq t_1.ts - t_2.ts \leq interval_2$.

(3) $t_1.ts > n_2 \cdot slide - interval_1$, where $n_2 = \lceil (t_2.ts - begin)/slide \rceil$

(4) $t_2.ts > n_1 \cdot slide - interval_2$, where $n_1 = \lceil (t_1.ts - begin)/slide \rceil$. □

Within Lemma 4.1, condition (2) basically says that the two joined tuples should appear in the corresponding window intervals. For instance, in Figure 2(a), tuple "14" from s_2 is the earliest tuple from s_2 that can be joined with tuple "20" from s_1. However, this condition alone cannot guarantee the correctness. For example, in Figure 2(b), tuple "23" from s_1 cannot join with tuple "22" from s_2 based on the window predicate definition. Condition (3) and (4) are used to deal with such cases. They ensure that there exists a pair of temporal relations at a particular time instance that contain the two tuples respectively.



s1 10 12 13 15 16 18 19 20
s2 10 11 13 14 15 16 17 19 20
(a) at τ = 20
s1 20 21 22 23 25 27 28 29 30
s2 20 22 23 24 25 27 28 30
(b) at τ = 30

Figure 2: Window join query Q between two streams s_1 and s_2, with $begin(Q) = 0$, $slide(Q) = 10$, $interval_1(Q) = 5$ and $interval_2(Q) = 7$. Only the timestamps of the arrived tuples are drawn.

Theorem 4.1. *A select-project-join (SPJ) continuous query Q_1 is contained by another SPJ continuous query Q_2 iff they satisfy either conditions (1) $\wedge$ (2) $\wedge$ (3) $\wedge$ (4) or conditions (1) $\wedge$ (2) $\wedge$ (3) $\wedge$ (5) $\wedge$ (6):*

(1) $Q_1^\infty \sqsubseteq Q_2^\infty$, where Q_i^∞ is a query resulted from setting all the window sizes of Q_i as ∞;

(2) $begin(Q_1) \geq begin(Q_2)$;

(3) $\forall i$, $inteval_i(Q_1) \leq interval_i(Q_2)$, where $interval_i(Q_j)$ is the window size of the ith stream involved in Q_j;

(4) $\forall i, 1 \leq slide(Q_2) \leq \max(1,\ interval_i(Q_2) - interval_i(Q_1))$, where $slide(Q_2)$ is the sliding step of Q_2.

(5) $\exists m \in [1, \infty)$, s.t. $begin(Q_2)+m\cdot slide(Q_2)-begin(Q_1) \leq \min_i(interval_i(Q_2) - interval_i(Q_1))$.

(6) $slide(Q_1) = k \cdot slide(Q_2)$, where k is a positive integer. □

The essential idea of Theorem 4.1 is, if Q_1 is contained by Q_2, then for every time instance τ_1 at which a temporal relation $\mathcal{R}_1^i(\tau_1)$ is defined for each stream by the window predicates in Q_1, there exists at least another time instance τ_2 at which a temporal relation $\mathcal{R}_2^i(\tau_2)$ is defined for each stream by the window predicates in Q_2 and $\mathcal{R}_2^i(\tau_2)$ contains $\mathcal{R}_1^i(\tau_1)$.

Conditions (1)-(3) are easy to understand. First, Q_2 has to contain Q_1 without considering the window predicates. Second Q_2 has to begin earlier than Q_1 and Q_2's window intervals should be as large as the corresponding ones in Q_1.

Condition (4) says that the sliding step of Q_2 is smaller than the difference of the two window intervals defined on the same stream. Figure 3 illustrates the reasoning behind this. It can be seen that the temporal relation defined by Q_2 at $\tau = 20$, $\mathcal{R}_2(18)$ (shown in Figure 3(b)) does not contain the one defined by Q_1, $\mathcal{R}_1(20)$ (shown in Figure 3(a). But, as long as the sliding step of Q_2 is smaller than the difference between the two window intervals, there would exist a

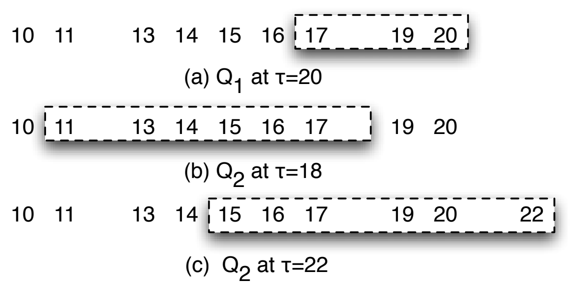


Figure 3: Temporal relations defined by two queries on the same stream. The parameters are $interval(Q_1) = 4, interval(Q_2) = 8$, and $slide(Q_2) = 4$.

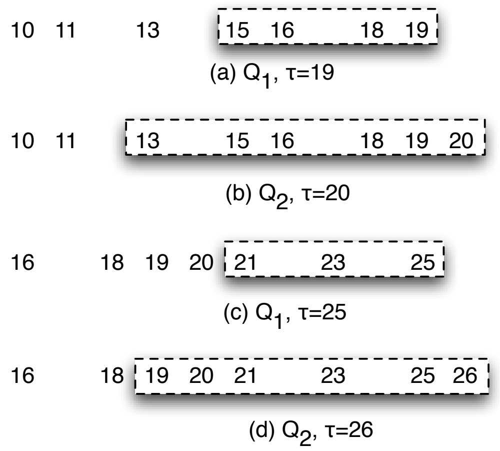


Figure 4: Temporal relations defined by two queries on the same stream. The parameters are $interval(Q_1) = 5, interval(Q_2) = 8, slide(Q_1) = 6$, and $slide(Q_2) = 3$.

time instance τ such that $\mathcal{R}_1(18) \subset \mathcal{R}_2(\tau)$. For example, Figure 3(c) shows that actually $\mathcal{R}_2(22)$ contains $\mathcal{R}_1(20)$.

Conditions (5) and (6) state the case that the windows of the two queries slide synchronously. Figure 4 shows an example. Here, $\mathcal{R}_2(20)$ contains $\mathcal{R}_1(19)$ as shown in Figure 4(a) and (b). Furthermore, as their sliding steps fulfill Condition (6), this containment pattern will repeat in the future time instances. Figure 4(c) and (d) illustrate the situation at another time instance.

Theorem 4.2. *A continuous stream query Q_1 with a generic aggregate function is contained by another continuous stream aggregate query Q_2 iff all the following conditions are true:*

(1) $Q_1^\infty \sqsubseteq Q_2^\infty$, where Q_i^∞ is a query resulted from setting all the window sizes of Q_i as ∞;

(2) $begin(Q_1) \geq begin(Q_2)$;

(3) $\forall i, interval_i(Q_1) = interval_i(Q_2)$, where $interval_i(Q_j)$ is the window size of the ith stream in query Q_j;

(4) $slide(Q_1) = k \cdot slide(Q_2)$ where k is a positive integer.

This reasoning of Theorem 4.2 is similar to Theorem 4.1. Hence, for brevity, we shall not reiterate here.

4.2 Query Merging Algorithms

Recall that, in our approach, the server maintains a number of query groups such that queries inside each group have overlapping results and it is beneficial to merge these queries into one representative query Q that contains all the member queries Q_i.

With the lemma and theorems developed in the previous section, we can generate the representative query for a group of queries as presented in the following subsections.

4.2.1 Merging SPJ Queries

The function to merge two SPJ queries is presented in Algorithm 1. It takes two queries as its input parameters and returns a query that contains them. In this paper, we only consider merging queries involving the same set of streams to avoid incurring large processing overhead. Line 3 enforces this constraint.

Algorithm 1: Merging two SPJ queries

```
1  MergeSPJ($(Q_1, Q_2)$)
2  begin
3      if $Strm(Q_1) \neq Strm(Q_2)$ then return error;
4      if $Q_1 \sqsubseteq Q_2$ then
5          $Q \leftarrow Q_2$;
6          $Attr(Q) \leftarrow Attr(Q_1) \cup Attr(Q_2)$;
7          if $Q_1 \neq Q_2$ then
8              $Attr(Q) \leftarrow Attr(Q) \cup$ attributes in $Pred(Q_1)$;
9      else if $Q_2 \sqsubset Q_1$ then
10         $Q \leftarrow Q_1$;
11         $Attr(Q) \leftarrow Attr(Q_1) \cup Attr(Q_2)$;
12         $Attr(Q) \leftarrow Attr(Q) \cup$ attributes in $Pred(Q_2)$ ;
13     else
14         $Strm(Q) \leftarrow Strm(Q_1)$;
15         $Pred(Q) \leftarrow Pred(Q_1) \cup Pred(Q_2)$;
16         $begin(Q) \leftarrow \min(begin(Q_1), begin(Q_2))$;
17         $slide(Q) \leftarrow \max(slide(Q_1), slide(Q_2))$;
18         $gcd \leftarrow$ GCD($slide(Q_1), slide(Q_2)$);
19         foreach $stream\ s_i \in Strm(Q)$ do
20             $diff \leftarrow interval_i(Q_1) - interval_i(Q_2)$;
21             $interval_i(Q) \leftarrow \max(interval_i(Q_1), interval_i(Q_2))$;
22             if $interval_i(Q_1) > interval(Q_2)$ then
23                 $s \leftarrow slide(Q_1)$;
24             else if $interval_i(Q_1) < interval(Q_2)$ then
25                 $s \leftarrow slide(Q_2)$;
26             else
27                 $s \leftarrow \min(slide(Q_2), slide(Q_1))$;
28             $slide \leftarrow \min(s, \max(gcd, diff))$;
29         $Attr(Q) \leftarrow Attr(Q_1) \cup Attr(Q_2)$;
30         $Attr(Q) \leftarrow Attr(Q) \cup$ attributes in $Pred(Q_1)$ or $Pred(Q_2)$;
31     return $Q$;
32 end
```

Lines 4-12 deal with the situation that one of the input queries Q_i contains the other Q_j. In this case, the function simply return Q_i. But note that the containment checking here does not consider which attributes are selected by the two queries. Therefore, we have to combine the attribute selection lists of both queries (line 29). Furthermore, to allow retrieval of the results of Q_j from that of Q_i, the attribute list Q is extended with those appear in $Pred(Q_j)$ (line 30)

Lines 13-28 perform the merging of two queries that do not contain each other They merge the predicates, stream windows and the selected attributes one after another. Among these, lines 19-28 refine the sliding step of Q step by step based on Theorem 4.1. Note that the merged predicates might be further reduced if some of them are covered by the others. Minimizing the number of predicates is a traditional NP hard problem, which attracts much attention [11, 23]. We will just adopt these results.

4.2.2 Merging aggregate queries

Based on Theorem 4.2, only only queries with the same input stream, the same Group By attributes and the same window intervals are considered for merging. The algorithm is straight-forward, so we leave it out of this paper.

4.3 Subscription Generation

As the result streams of only the representative queries will be delivered over a DPSS, users have to register subscriptions to the DPSS to retrieve the results that are of interest to them. This subsection presents how to generate such subscriptions.

Suppose Q is the representative query for a query group and Q_i is one of the members of this group. The algorithm first initialize the subscription with the Q's result stream name, and then it adds the selected attribute list and predicates of Q_i. After that, filters are added to check whether the result tuples satisfy the window predicate defined by Q_i. These filters are generated based on Lemma 4.1. Again the filters here could be further optimized, which is out of the scope of this paper.

Algorithm 2: Subscription Generation

```
1  SubGen($(Q, Q_i)$)
2  begin
3      $Sub.\mathcal{S} \leftarrow \{Result(Q)\}$;
4      $Sub.\mathcal{P} \leftarrow Attr(Q_i)$;
5      $Sub.\mathcal{F} \leftarrow Pred(Q_i)$;
6      $n \leftarrow$ the number of stream involved in $Q_i$;
7      $bg \leftarrow begin(Q_i)$;
8      $sl \leftarrow slide(Q_i)$;
9      for $j = 1; j < n; j++$ do
10         for $k = j+1; k <= n; k++$ do
11             $Sub.\mathcal{F} \leftarrow Sub.\mathcal{F} \wedge$
12             $(-1 \cdot inv[j] \leq ts_j - ts_k \leq inv[k]) \wedge$
13             $ts_j > \lceil (ts_k - bg)/sl \rceil * sl - inv[j] \wedge$
14             $ts_k > \lceil (ts_j - bg)/sl \rceil * sl - inv[k]$;
15     return $Sub$;
16 end
```

5. QUERY GROUPING

With the above algorithms, one can merge a query group and efficiently deliver the query results to the individual users. This section investigates how to partition the queries into multiple groups. In particular, we study the optimization algorithms and maintenance mechanisms of query grouping under the context that queries are frequently inserted and terminated.

5.1 Benefit Estimation

The first problem of optimizing query grouping is how to estimate the benefit of merging a query group. The benefit considered in this paper is the amount of data communication overhead that can be saved. A common cost metric is adopted here: $\sum_i l_i \cdot c_i$, where l_i is the transmission latency of the ith link in the overlay network of the DPSS and c_i is the communication traffic per unit time on l_i.

To accurately estimate the benefit of merging a query group, one can count the data transfer rate on each overlay link if we know the exact data dissemination tree. Unfortunately, in a large scale network, it is hard to maintain information in such a detail.

As the problem of how to maintain network structure knowledge in a scalable way is out of the scope of this paper, we only adopt a cost model assuming little knowledge of the network. Since the actual cost is counted in experimental results, this actually biases against our method. Moreover, if more network knowledge exists, the cost model can be replaced with a more accurate one without much change of our algorithms.

More specifically, the stream processing server only keeps track of the next hop of the delivery path of the result streams. The benefit of the query merging is estimated as $(\sum_i C(Q_i) - C(Q)) \cdot l$, where $C(Q)$ is the data rate (bits/sec) of the representative query Q's result stream, while $C(Q_i)$ is the data rate of the member query Q_i's result stream. Furthermore l is the latency of the common first hop of all the member query Q_i. This implicitly says that only queries with a common first hop would be merged, which is intuitive.

The estimation of result stream rate is a common task required by most query optimizers. Therefore, existing techniques in stream query optimization [5] can be used for this purpose. Furthermore, the data statistics required by our cost model can be shared with the query optimizer and hence little extra overhead will be incurred to maintain the statistics.

5.2 Query Groups Maintenance

There are a few challenges of the query grouping and our system addresses them in the following ways:

1. Achieve high benefit. The benefit estimation function discussed above is used to estimate the benefit of a grouping. Heuristics are required to derive a good grouping.

2. Incremental grouping. In reality, queries often come one at a time and should be started running at its submission. As there is no way to control the order of queries that come to system, grouping has to be done incrementally.

3. Efficient grouping maintenance. We consider the situation that queries would be inserted and terminated anytime. Hence, it is necessary to have an efficient data structure to maintain the query grouping and to facilitate the decision making of when and how to re-merge queries. In this paper, a multi-tree data structure is adopted to serve this purpose. In this structure, a query tree is built for each query group. The root of the tree is the representative query of the query group. Within the tree, a query Q_i is an ancestor of another one Q_j only if Q_i contains Q_j. Figure 5 shows an example of such a tree.

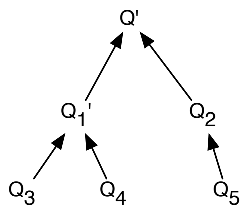


Figure 5: An example query tree. This tree represents a group of query: Q_2, Q_3, Q_4 and Q_5. Q'_1 is a derived query constructed by merging Q_3 and Q_4. Q_5 is contained by Q_2. Q' is the representative query of the whole query group which is derived by merging Q'_1 and Q_2.

As we will see later, such a query tree is helpful in quickly determining whether the merging of the queries under each subtree is beneficial and hence whether re-optimization is required. Furthermore, if some queries in a subtree is terminated, the synthetic queries along the path from the terminated queries to the root might be rewritten, while the other subtrees need not be modified.

4. Re-placing queries into different groups would change the user subscriptions and hence may incur many message exchanges among the nodes in the network in order to modify the routing table in the DPSS. Hence, we try to avoid frequent migration of a query from one group to another. Query grouping is maintained periodically so that the frequency of the changes can be regulated by the length of the period. Furthermore, a query will be retained in the same group as long as it is still beneficial to do so.

More details of the algorithms are presented in the rest of this section.

5.2.1 New query insertion

As mentioned, the ordering of query arrival is in general out of control and query grouping has to be done as each query arrives at the system. This section describes procedure of inserting a new query to the query groups.

When a query is first submitted to the system, a query group is selected by using a greedy algorithm (Algorithm 3, which estimates the benefit of merging the new query with each existing query group and select the one with the highest benefit. If no merging has positive benefit, then a new group is generated. Then the new query is added to the corresponding query tree structure of the selected query group or the newly generated group. If the existing representative query of the group and the new query do not contain one another, a new representative query is generated for this query group.

As a side note, query merging may incur query processing overhead. To avoid getting arbitrary high overhead, a threshold can be used to restrict how much overhead can be accepted to trade for the communication efficiency. Here, we use a threshold parameter α, which is the maximum percentage of processing overhead that can be accepted. For example, if $\alpha = 0.1$, then the system can tolerate 10% of processing overhead incurred by query merging. If the merging incurs the overhead higher than this threshold, then it will

Algorithm 3: Query Insertion

```
1  Insert($newQuery$)
2  begin
3      $max \leftarrow 0$; $toMerge \leftarrow null$;
4      foreach $rootQ \in trees$ do
5          $bf \leftarrow$ benefit of merging $rootQ$ with $newQuery$;
6          $oh \leftarrow$ processing overhead incurred by merging
           $rootQ$ with $newQuery$ + the current overhead
           of the query group of $rootQ$;
7          if $bf > max$ & $oh < \alpha$ then
8              $max \leftarrow bf$;
9              $toMerge \leftarrow rootQ$;
10     if $toMerge = null$ then
11         $trees$.addRoot($newQuery$);
12     else if $newQuery \sqsubseteq toMerge$ then
13         AddChild($toMerge, newQuery$);
14     else if $toMerge \sqsubseteq newQuery$ then
15         AddChild($newQuery, toMerge$);
16         replace $toMerge$ with $newQuery$ in $trees$
17     else
18         $newRoot \leftarrow$ MergeQ($toMerge, newQuery$);
19         add both $toMerge$ and $newQuery$ to
           $newRoot.childlist$;
20         replace $toMerge$ with $newRoot$ in $trees$;
21 end
22 AddChild($parent, newChild$)
23 begin
24     foreach $child \in parent.childlist$ do
25         if $newQuery \sqsubseteq child$ then
26             AddChild($child, newQuery$);
27             return;
28     add $newChild$ to $parent.childlist$;
29 end
```

not be considered. In the estimation of the processing cost, again existing stream query optimization techniques can be used, such as [5].

5.2.2 Query termination

When a query terminates, Algorithm 4 is run to modify the query trees to reflect the changes. Two types of queries are distinguished in the algorithm: (1) original queries: those queries submitted by the users; (2) derived queries: those queries derived from query merging.

First, the to-be-terminated query is removed from the tree and then transfer its children to its parent or generate a new root if the to-be-terminated query is a root itself.

Second, if the to-be-terminated query's parent is a derived query, the merging algorithm will be run to rewrite the parent query to reflect the change. Rewriting will be propagated up in the tree till the node which is unnecessary to be rewritten. Note that the rewriting of these synthetic queries will not incur changes on the network side (i.e. the subscriptions of the users can remain unchanged). Instead, the rewriting can reduce the communication cost by the possible "tightening" of the representative queries. Hence, we choose to perform this eagerly.

On the other hand, if a query in a query group is terminated, then it might not be beneficial for other queries to be placed in this query group any more. For example, a query is placed into this group because of its overlap with the terminated query. Now, it might not be beneficial to keep it in this group. The grouping could be re-optimized. However, as we have discussed, moving the query from one group to another has to change the subscription of the user which will incur changes on the overlay network, i.e. the change of user subscriptions and hence the routing tables.

Therefore, a lazy approach is adopted for the re-optimization of grouping. The re-optimization algorithm is only run periodically, which will be presented in the next subsection. The query termination algorithm only marks the re-written derived queries for re-optimization, which will be done at the coming re-optimization round.

Algorithm 4: Query Termination

```
1  Terminate($q$)
2  begin
3      remove $q$ from its query tree;
4      if $q.childlist \neq null$ then
5          if $q.parent \neq null$ then
6              $q.parent.childlist.add(q.childlist)$;
7          else
8              $newQuery \leftarrow$ merge all the child queries of $q$;
9              $newQuery.isDerived \leftarrow true$;
10             $newQuery.to\_reoptimize \leftarrow true$;
11     if $q.parent \neq null$ & $q.parent.isDerived$ then
12         Rewrite ($q.parent$);
13         $q.parent.to\_reoptimize \leftarrow true$;
14 end
15 Rewrite ($q$)
16 begin
17     Merge all the child queries of $q$ to a new query
       $newQ$;
18     if $q$ is not semantically equivalent to $newQ$ then
19         $q \leftarrow newQ$;
20         if $q.parent \neq null$ & $q.parent.isDerived$ then
21             Rewrite($q.parent$);
22 end
```

5.2.3 Query group re-optimization

Periodically, Algorithm 5 will be run to re-optimize the query grouping. In line 4, the algorithm traverses the query trees and re-optimize them one by one. After that, it gets a list of queries that should be considered to be replaced into different groups. Then line 5 uses the above query insertion algorithm to replace the query one by one.

The re-optimization algorithm for each query group is shown in Lines 7-22. This algorithm takes an inputs *queryNode* (a node in a query tree) and inserts into *toReplace* the queries that are currently in the substree rooted at *queryNode* but are no longer beneficial be grouped with other queries in the subtree. It is done by traversing the query tree in depth-first order and recursively calling the algorithm on each node in the tree.

For each node, after calling the algorithm recursively on all the child nodes, the algorithm gets a list of query nodes,

Algorithm 5: Query Group Re-Optimization

```
1  ReopitmizeGroups()
2  begin
3      $toReplace \leftarrow \emptyset$;
4      foreach $root \in trees$ do
           Reoptmize($root, toReplace$);
5      foreach $query \in toReplace$ do Insert($query$);
6  end
7  Reoptmize($queryNode, toReplace$)
8  begin
9      $newChildlist \leftarrow \emptyset$;
10     foreach $child \in queryNode.childlist$ do
11         Reoptmize($child, newChildlist$);
12     if $queryNode.to\_reoptimize$ then
13         $b \leftarrow$ benefit of merging all queries in
           $\{queryNode.childlist \cup newChildlist\}$;
14         if $b \leq 0$ then
15             $toReplace.add(queryNode.childlist)$;
16             $toReplace.add(newChildlist)$;
17             remove $q$ from the trees;
18         else
19             $queryNode \leftarrow$ merge all queries in
               $\{queryNode.childlist \cup newChildlist\}$;
20         if
           $toReplace \neq \emptyset$ & $queryNode.parent.isDerived$
           then
21             $queryNode.parent.to\_reoptimize \leftarrow true$;
22 end
```

newChildlist, which are the queries that are no longer be beneficial to be placed in the subtrees of the individual child nodes. However, within these queries, those extracted from the subtree of one child node may still have overlap with the queries in the subtree of another child node. Hence, lines 12-19 check whether it is beneficial to merge the queries in *newChildlist* together with its current children. If so, then it simply performs the merging and add all the nodes in *newChildlist* to the current nodes childlist. Otherwise, it returns all these nodes to its parent node.

Note that to minimize the overhead of revising the user subscriptions that would be incurred by re-grouping, queries would be considered for re-grouping only when the current grouping has negative benefit (line 14). Using a threshold here might be able to get a better trade-off. Unfortunately, by our experiments, this cannot achieve significant benefit. A high positive threshold does not work well, because the more overlap that the query has with the current group, the less benefit that will be achieved by re-grouping the query. On the other hand, a high negative threshold does not work well too. The reason is, with a negative benefit, the query already has no overlap with the current query group and, in general, the benefit of re-grouping such a query would dominate the user subscription update cost. Furthermore, by experiments, there is no significant difference between a small positive/negative threshold and 0. But the performance degenerates quickly with a slightly higher positive/negative threshold. Therefore, such a threshold is not considered in this paper.

6. PERFORMANCE STUDY

Implementation. The algorithms in this paper are implemented in a middleware on top of our stream processing system: GSN (Global Sensor Network, `http://gsn.sourceforge.net/`) [1], which is tailored for efficient processing of sensor data and managing the connections with various heterogeneous sensor networks. The system is implemented mainly in Java. The experiment is conducted in a Linux server with 2 Dual-Core 2.66GHz Intel CPU and 4G memory.

Data set. We use the sensor data set collected by our SensorScope project (`http://sensorscope.epfl.ch`), which measures key environmental data such as air temperature and humidity, surface temperature, incoming solar radiation, wind speed and direction, precipitation, and soil moisture and pressure. The data from each sensor are treated as one data stream. In the experiments, we use 63 streams as our data set and emulate the streaming scenario by using their timestamp information.

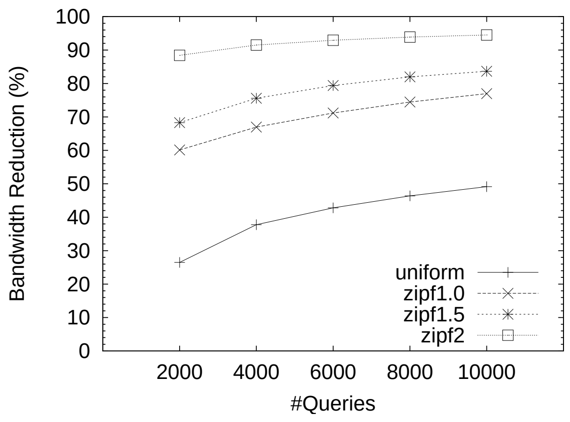
Query generation. Each query in the experiment is generated randomly in the way described as follows. First, a random number of streams (uniformly distributed between 1 and 5) are selected randomly to be involved in the query. Then a random number of predicates (uniformly distributed between 1 and 5) are generated based on the column information of the streams (such as the column names, the maximum/min values etc.). In the experiments, we vary the distribution used to select the streams and the portion of data selected by the predicates. Both uniform and zipfian distribution are used. Furthermore, the window predicates are generated with random parameters (time intervals, sliding steps and starting times). Finally, the projection attributes and aggregate functions are generated randomly. All the experiments are repeated 20 times with different random queries and the average results are reported.

DPSS. The DPSS which is used to disseminate the query results is simulated in the experiments. The topology generator BRITE (`http://www.cs.bu.edu/brite/`) is used to generate a power law network topology with 1000 nodes. Then a minimum spanning tree is constructed as the dissemination tree. One of the nodes is selected as the stream query processor and, for each query, a random node is selected as the origin of the query, which should be the destination of the query result.

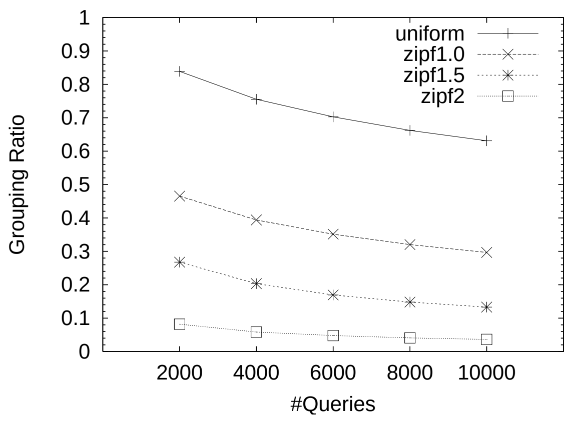
6.1 Query Insertion

In this subsection, we examine the performance of query insertion. In the experiments, queries arrive at the system one by one. Our query insertion algorithm is run to optimize the query grouping incrementally.

In the first experiment, we set the overhead threshold α at a relatively high value (0.5) to see how much benefit we can get without worrying too much about the processing overhead. In Figure 6(a), we present the *bandwidth reduction* at each time instance when a certain number of queries are inserted. *Bandwidth reduction* is computed as the percentage of the sum of the bandwidth consumption of each overlay link (weighted by the latency of each link as discussed in Section 5.1) that is reduced by the query merging in comparison to the case without merging. A few interesting points can be identified from the figure. First, with more number of queries added to the system, there are more opportunities

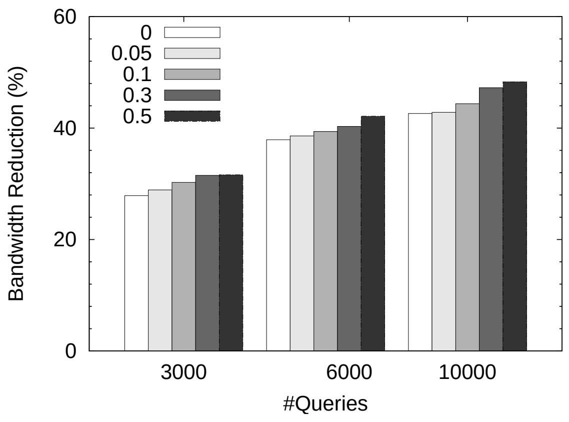


(a) Bandwidth Reduction

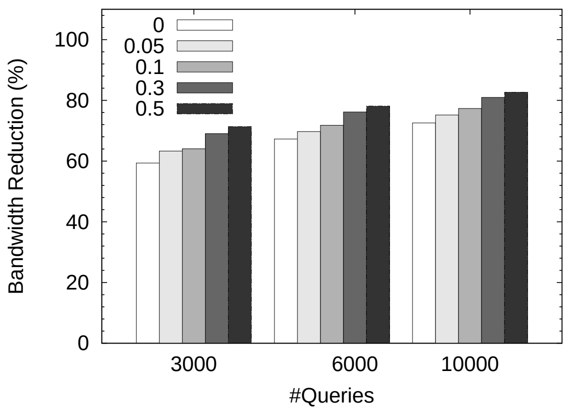


(b) Grouping Ratio

Figure 6: Query Insertion



(a) Uniform



(b) Zipf 1.5

Figure 7: Sensitivity to α

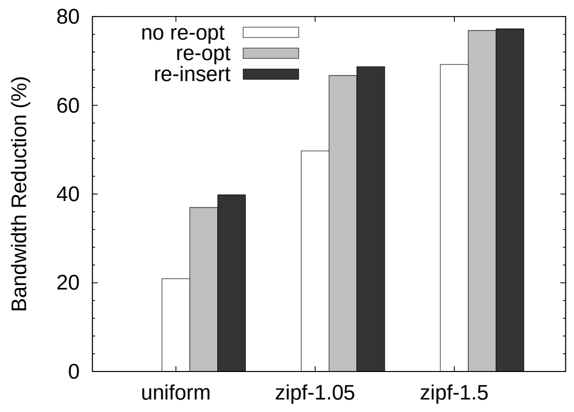
for the query merging approach to explore the sharing of communication and hence a larger bandwidth reduction can be achieved. Another interesting point is that query merging is more beneficial with a skewed query distribution. The reason is obvious. With more queries interested in the same subset of data, the probability that we can merge the queries would be higher. Figure 6(b) provides another perspective on the experimental results. The *grouping ratio* is the ratio of the number of query groups to the total number of queries. Generally, the lower the grouping ratio, the higher the bandwidth reduction could be.

Another experiment is to investigate the sensitivity of our algorithms to the processing overhead threshold α. Figure 7 shows the result of the experiments. The value of α (in the query insertion algorithm under Section 5.2.1) is varied from 0 to 0.5. The general trend is, a higher α value provides more opportunities for query merging and hence the outcome bandwidth reduction is higher. Note that the difference is not very significant. The reason could be, for the randomly generated query set in this experimental study, only a small number of merging could incur high processing overhead. We have varied the query generation parameters but could not find a set of parameters that can make this difference more significant. Hence in such query set, a very low α is much desirable, as it can significantly reduces the communication cost without incurring much processing overhead. In reality, there could exist some query set that could be more sensitive to the value of α. The tuning of an optimal α value could get a better trade-off between communication cost and processing cost.

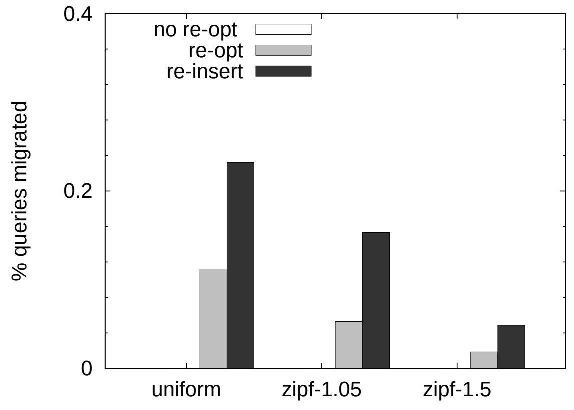
6.2 Query Grouping Re-optimization

In this section, we examine the performance of the query re-optimization algorithms. In the experiment, $10,000$ queries are first inserted into the system and then we terminate half of the queries. It is compared with two cases: (1) without running the re-optimization algorithm ("no re-opt") and (2) running the insertion algorithm on all the remaining queries from scratch ("re-insert").

Figure 8(a) shows the comparison of the bandwidth reduction among the three cases and Figure 8(b) presents the percentage of queries that have been migrated to another query group. As one can see, "Re-opt" can achieve much larger bandwidth reduction than the case without re-optimization. Furthermore, "re-insert" works slightly better than "re-opt". This is because "re-insert" can explore a larger solution space than "re-opt". However, as shown in Figure 8(b), "re-insert" incurs much more migrations than "re-opt", which would result in higher overhead over the

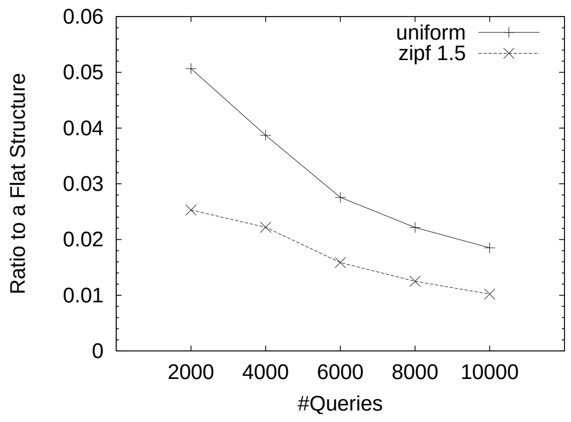


(a) Bandwidth reduction

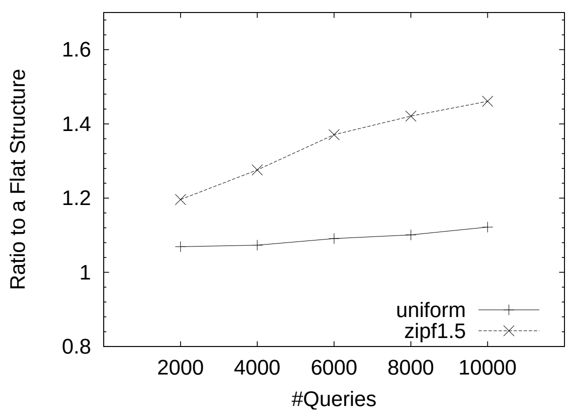


(b) %queries migrated

Figure 8: Query Grouping Re-optimization



(a) Time to terminate a query



(b) Time to insert a query

Figure 9: Tree structure vs. flat structure

network.

Another interesting point that can be observed is, with a more skewed query distribution, less benefit can be achieved by re-optimizing the query grouping. This is because more queries have overlap relationships and hence less queries need be migrated to another query group. It is also reflected in Figure 8(b). Both "re-opt" and "re-insert" migrate less queries for a more skewed query distribution.

6.3 Efficiency of the Query Tree

This experiment is to examine whether the query tree is effective to enhance the query grouping maintenance efficiency. We compare it with a flat structure, where queries within each group are kept in a flat list. We compare the maximum query termination and query insertion time between the two approaches. In the experiment, we first insert a certain number of queries into the system and then try to insert (or remove) a query to (or from) the most popular query group. This is expected to be the maximum insertion (or termination) time. One can see from Figure 9(a), the tree approach is much more efficient in query termination than the flat one. That is because it avoids the running of unnecessary query merging while a flat structure cannot exploit this opportunity. The difference is more obvious with a skewed query distribution and a larger query population where query groups tend to have more members. On the other hand, it can be seen from Figure 9(b) that the tree approach works slightly worse than the flat structure for query insertion. This is due to the fact that, in the tree approach, a new query has to travel a few levels down the query tree before it is settled in a place within the tree. But this is not necessary in a flat structure. In summary, for systems with similar query termination and insertion rates, the query tree approach is much more efficient than the flat one.

7. CONCLUSION

This paper addresses an important stream processing issue that we faced during our deployments, query result stream delivery, which is often overlooked by existing stream processing systems, and proposes an easy-to-implement yet effective solution. To enhance the system's scalability, DPSS, a scalable and efficient communication paradigm, is employed to deliver query result streams. To fully exploit the message delivery sharing capability of a DPSS, we propose a query grouping and merging approach. To realize this approach, stream query containment theorems are first studied, based on which, query merging and query grouping algorithms are proposed. To deal with the frequent arrival and removal of queries, a multi-tree structure is used to facilitate efficient maintenance of query grouping. Furthermore, adaptive re-optimization algorithms are proposed to continuously adapt the query grouping to the change of the query set and meanwhile keep the query migration overhead to be low. The experiments conducted show that this approach is very efficient and effective , especially with a large number of queries or a skewed query distribution.

Based on the techniques proposed by this paper, there are a few interesting problems to explore in the future. First, the current approach merges all the queries in a query group into one representative query. However, sometimes this would increase the processing cost. Another possible approach is to generate one or more representative queries for each query group and keep the processing overhead as low as possible. User subscriptions then should subscribe to multiple result streams instead of one. Second, in a distributed stream processing system, query operators can be allocated to multiple processing servers. It is interesting to study how the operator allocation and the query grouping and merging will interact with each other.

8. REFERENCES

[1] K. Aberer, M. Hauswirth and A. Salehi. Infrastructure for data processing in large-scale interconnected sensor networks, In *MDM*, 2007.

[2] M. K. Aguilera, R. E. Strom et al. Matching events in a content-based subscription system. In *PODC*, pages 53–61, 1999.

[3] Y. Ahmad and U. Çetintemel. Networked query processing for distributed stream-based applications. In *VLDB*, pages 456–467, 2004.

[4] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, pages 336–347, 2004.

[5] A. Ayad and J. F. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In *SIGMOD*, pages 419–430, 2004.

[6] B. Babcock et al. Load Shedding for Aggregation Queries over Data Streams. In *ICDE*, pages 350–361, 2004

[7] D. Calvanese, G. D. Giacomo, and M. Lenzerini. On the decidability of query containment under constraints. In *PODS*, pages 149–158, 1998.

[8] A. Carzaniga et al. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[9] A. Carzaniga, M. Rutherford, and A. L. Wolf. A routing scheme for content-based networking. In *INFOCOM*, 2004.

[10] A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In *SIGCOMM*, 2003.

[11] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90, 1977.

[12] B. Chandramouli, J. Xie, and J. Yang. On the database/network interface in large-scale publish/subscribe systems. In *SIGMOD*, 2006.

[13] B. Chandramouli and J. Yang. End-to-End support for joins in large-scale publish/subscribe systems In *VLDB*, 2008.

[14] J. Chen, D. J. DeWitt, F. Tian, Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD*, pages 379–390, 2000.

[15] S. Cohen, W. Nutt, and Y. Sagiv. Containment of aggregate queries. In *ICDT*, pages 111–125, 2003.

[16] A. Crespo, O. Buyukkokten, and H. Garcia-Molina. Query Merging: Improving Query Subscription Processing in a Multicast Environment. In *IEEE Trans. Knowl. Data Eng.*, 15(1):174–191, 2003.

[17] Y. Diao, S. Rizvi, and M. J. Franklin. Towards an Internet-Scale XML Dissemination Service. In *VLDB*, 2004.

[18] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.

[19] H. Jafarpour, B. Hore, S. Mehrotra, and N. Venkatasubramanian. Subscription Subsumption Evaluation for Content-Based Publish/Subscribe Systems. In *Middleware*, pages 62–81, 2008.

[20] S. Kirkpatrick et al. Optimization by simulated annealing. Science, Number 4598, 1983.

[21] P. G. Kolaitis, D. L. Martin, and M. N. Thakur. On the complexity of the containment problem for conjunctive queries with built-in predicates. In *PODS*, pages 197–204, 1998.

[22] S. Krishnamurthy, C. Wu, and M. J. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, 2006.

[23] I. Kunen and D. Suciu. A scalable algorithm for query minimization. Technical report. U. Washington, 02-11-04, 2002.

[24] S. Madden et al. Continuously adaptive continuous queries over streams. In *SIGMOD* , pages 49–60, 2002.

[25] S. Michel et al. Environmental Monitoring 2.0. In *ICDE* , 2009.

[26] W. Nutt, Y. Sagiv, and S. Shurin. Deciding equivalences among aggregate queries. In *PODS*, pages 214–223, 1998.

[27] A. M. Ouksel, O. Jurca, I. Podnarx, and K. Aberer, Efficient Probabilistic Subsumption Checking for Content-Based Publish/Subscribe Systems. In *Middleware*, pages 121–140, 2006.

[28] O. Papaemmanouil et al. Semcast: Semantic multicast for content-based data dissemination. In *ICDE*, pages 242–253, 2005.

[29] Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *J. ACM*, 27(4):633–655, 1980.

[30] N. Tatbul et al. Load Shedding in a Data Stream Manager. In *VLDB*, pages 309–320, 2003.

[31] The STREAM Group. STREAM: The stanford stream data manager. *IEEE Data Engineering Bulletin.*

[32] Y. Xing, S. B. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *ICDE*, pages 791–802, 2005.

[33] R. Zhang, N. Koudas, B. C. Ooi, D. Srivastava. Multiple Aggregations Over Data Streams. In *SIGMOD*, pages 299–310, 2005.

[34] Y. Zhou, K. Aberer and K.-L. Tan. Toward Massive Query Optimization in Large-Scale Distributed Stream Systems. In *Middleware*, 2008.